Agile Model Driven Development (AMDD)

Scott W. Ambler
Practice Leader Agile Development, IBM

Modeling is an important part of all software development projects because it enables you to think through complex issues before you attempt to address them via code. This is true for agile projects, for not-so-agile projects, for embedded projects, and for business application projects. Unfortunately, many modeling efforts prove to be dysfunctional. At one end of the spectrum are projects where no modeling is performed, either because the developers haven't any modeling skills or because they have abandoned modeling as a useless endeavor. At the other end of the spectrum are projects which sink in a morass of documentation and overly detailed models, either because the project team suffers from "analysis paralysis" and finds itself unable to move forward or because the team has burdened itself with too many modeling specialists who don't have the skills to move forward even if they wanted to. Somewhere in the middle are project teams that invest in modeling and documentation efforts only to discover that the programmers ignore the models anyway, often because the models are unrealistic or simply because the programmers think they know better than the modelers (and often they do). The goal of Agile Model Driven Development (AMDD) is to show how to avoid these problems, to gain the benefits of modeling and documentation without suffering the drawbacks.

1 Agile Models

A *model* is an abstraction that describes one or more aspects of a problem or a potential solution addressing a problem. Traditionally, models are thought of as zero or more diagrams plus any corresponding documentation. However non-visual artifacts such as use cases, a textual description of one or more business rules, or a collection of class responsibility collaborator (CRC) cards [1] are also models. An *agile model* [2] is a model that is just barely good enough [18] for the situation at hand. Agile models are just barely good enough when they exhibit the following traits:

- Agile models fulfill their purpose.
- Agile models are understandable.
- Agile models are sufficiently accurate.
- Agile models are sufficiently consistent.
- Agile models are sufficiently detailed.
- Agile models provide positive value.
- Agile models are as simple as possible.

Figures 1 and 2 both depict agile models. Figure 1 depicts a hand-drawn architecture sketch for a business application which was created by the team on the first few days of the project. Figure 2 depicts a physical data model (PDM) using the Unified Modeling Language (UML) notation [3] – it is possible to data model effectively using the UML. Both models are agile even though they're very different from each other:

- The data model is very likely a keeper whereas the sketch would be discarded once it's served its purpose.
- The data model was created using a sophisticated modeling tool whereas the sketch was created using very simple tool.
- The data model was created using a sophisticated notation, yet the sketch is clearly free-form.

- The data model depicts technical, detailed design whereas the sketch is high-level.

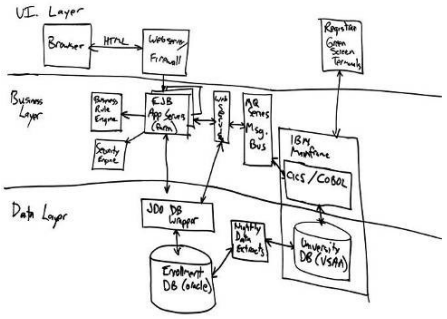

Figure 1: A hand-drawn architectural sketch.

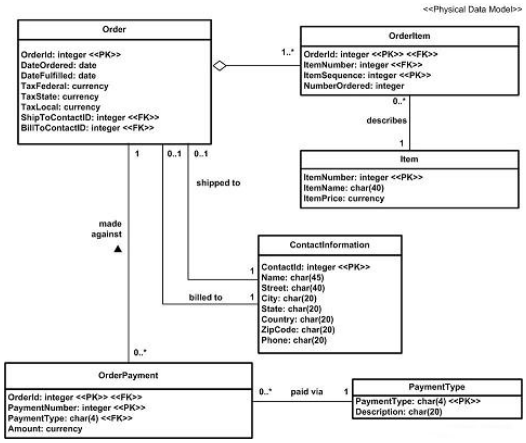

Figure 2: A physical data model (PDM).

One of the more controversial concepts in Agile Modeling is that agile models and agile documents are sufficient for the task at hand, or as I like to say they are "just barely good enough". For some reason people think that just barely good enough implies that the artifact isn't very good, when in fact nothing could be further from the truth. When you stop and think about it, if an artifact is just barely good enough then by definition it is at the most effective point that it could possibly be at. Figure 3 summarizes the value curve for an artifact being just barely good enough. Value refers to the net benefit of the artifact, which would be calculated as benefit - cost. The dashed line is at the point where the artifact is just barely good enough: anything to the left of the line implies that you still have work to do, anything to the right implies that you've done too much work. When you are working on something and it isn't yet barely good enough then you can still invest more effort in it and gain benefit from doing so (assuming of course you actually do work that brings the artifact closer to it's intended purpose). However, if an artifact is already just barely good enough (or better) then doing more work on it is clearly a waste: once an artifact fulfils its intended purpose then any more investment in it is useless bureaucracy. The diagram is a little naive because it is clearly possible for the value to be negative before the artifact becomes barely good enough although for the sake of argument I'm going to assume that you do a good job right from the beginning.

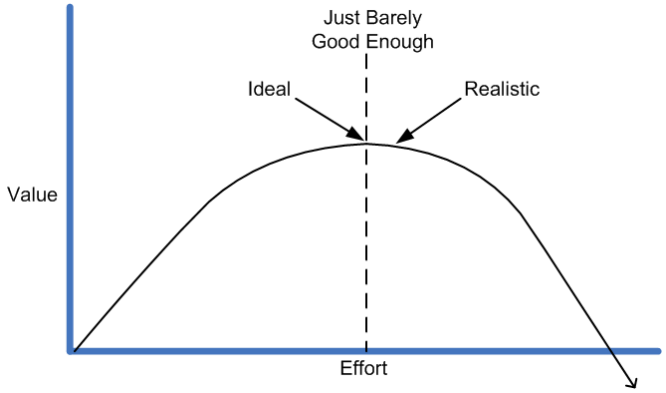

Figure 3: Just barely good enough is the most efficient.

The fundamental challenge with "just barely good enough" is that it is situational. For example, I often draw UML Sequence diagrams on a whiteboard to explore complex logic and then discard it once I'm done with it. In this case the whiteboard diagram is fine because it helps me to solve the issue which I'm thinking through with whomever I'm working. But, what if we're in a situation where we'll need to update this logic later on AND will want to do it via the diagram instead of via source code? Clearly a hand-drawn sketch isn't good enough in this case and we'll want to create a detailed diagram using a sophisticated CASE tool. We'd still create an agile model even though it is much more sophisticated than a sketch because "just barely good enough" reflects the needs of the situation.

It is important to distinguish between the orthogonal concepts of models and documents: some models become documents, or parts of documents, although many models are discarded after they have been used. I suspect that 90% or more of all models are discarded – how many whiteboard sketches have you erased throughout your career? For the sake of definition a document is a permanent record of information, and an *agile document* [2] is a document that is just barely good enough. The principles and practices of Agile Modeling, described in the next section, are applicable to both modeling and documentation.

2. Agile Modeling (AM)

The Agile Modeling (AM) method defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. These practices can be used to extend agile processes such as Extreme Programming (XP) [4], Feature Driven Development (FDD) [5], and Rational Unified Process (RUP) [6]. AM is a chaordic collection of practices – guided by principles and values – that should be applied by software professionals on a day-to-day basis. The focus of AM is to make your modeling and documentation efforts lean and effective; AM does not address the complete system lifecycle and thus should be characterized as a partial process/process. The advantage of this approach is that organizations may benefit from the focused guidance of a partial process. The disadvantages are that organizations need the requisite knowledge and skills to know which processes exist and how to combine them effectively. The concept of partial processes seems strange at first, but when you reflect a bit you quickly realize that partial processes are the norm – development processes, such as XP and the RUP, address the system development lifecycle but do not address the full IT lifecycle. The Enterprise Unified Process (EUP) [7] – an extension to the RUP which addresses the production and retirement phases of a system, operations and support

of a system, and cross-system issues such as enterprise architecture and strategic reuse – represents a full IT lifecycle.

AM is practices-based, it is not prescriptive. In other words it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeler. The advantage of describing a process as a collection of practices is that it is easy for experienced professionals to learn and reflects (hopefully) what they actually do, the disadvantage is that it does not provide the detailed guidance for novices. Prescriptive processes, on the other hand, often provide the detailed guidance required by novices but are ignored by experienced professionals. Prescriptive processes are well suited as training material for new hires and perhaps as input into process audits to fulfill the requirement that you have a well documented process.

Think of AM as more of an art than a science. It is defined as a collection of values, principles, and practices. (www.agilemodeling.com/values.htm), (www.agilemodeling.com/principles.htm), (www.agilemodeling.com/practices.htm). The values of AM include those of XP v1 [20]– *communication*, *simplicity*, *feedback*, and *courage* – and extend it with *humility* (XP v2 [21] adds the fifth value of *respect*, which I argue comes from *humility*). The principles of AM, many of which are adopted or modified from XP, provide guidance to agile developers who wish to be effective at modeling and documentation. They provide a philosophical foundation from which AM's practices are derived. The practices of AM are what people actually do. There is not a specific ordering to the practices, nor are there detailed steps to complete each one – you simply do the right thing at the right time.

Because every project team is different, and every environment is different, you should tailor your process to reflect your situation. AM reflects this philosophy – to claim that you are "doing AM" you merely need to adopt its values, its core principles and practices (see Table 1). The remaining principles and practices are optional, although they are very good ideas and should be adopted whenever possible. This approach enables you to tailor AM to meet your exact needs. Table 2 lists the supplementary principles and practices although for brevity does not describe them in detail.

Why would you want to adopt AM? AM defines and shows how to take a light-weight approach to modeling and documentation. What makes AM a catalyst for improvement is not the modeling techniques themselves – such as use case models, class models, data models, or user interface models – but how to apply them productively. Although you must be following an agile software process to truly be agile modeling, you may still adopt and benefit from many of AM's practices on non-agile projects.

Table 1. The core principles and practices of AM.

Core Principles	Core Practices
• Assume Simplicity • Embrace Change • Enabling the Next Effort is Your Secondary Goal • Incremental Change • Maximize Stakeholder Investment • Model With a Purpose • Multiple Models • Quality Work • Rapid Feedback • Software is Your Primary Goal • Travel Light	• Active Stakeholder Participation • Apply the Right Artifact(s) • Collective Ownership • Single Source Information • Create Several Models in Parallel • Create Simple Content • Depict Models Simply • Display Models Publicly • Iterate To Another Artifact • Model in Small Increments • Model With Others • Prove it With Code • Use the Simplest Tools

Table 2. Supplementary principles and practices.

Supplementary Principles	Supplementary Practices
• Content is more important than representation • Open and honest communication • Work with people's instincts	• Apply modeling standards • Apply patterns gently • Discard temporary models • Formalize contract models • Update only when it hurts

3. Agile Model Driven Development (AMDD)

As the name implies, AMDD is the agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. A primary driver of MDD is the Object Management Group (OMG)'s Model Driven Architecture (MDA) standard [11]. With MDD the goal is typically to create comprehensive models, and then ideally generate software from those models. This is a great vision, but one that may not be possible for all development teams.

AMDD takes a much more realistic approach: its goal is to describe how developers and stakeholders can work together cooperatively to create models which are just barely good enough. It assumes that each individual has some modeling skills, or at least some domain knowledge, that they will apply together in a team in order to get the job done. It is reasonable to assume that developers will understand a handful of the modeling techniques out there, but not all of them. It is also reasonable to assume that people are willing to learn new techniques over time, often by working with someone else that already has those skills. AMDD does not require everyone to be a modeling expert, it just requires them to be willing to try. AMDD also allows people to use the most appropriate modeling tool for the job, often very simple tools such as whiteboards or paper, because you want to find ways to communicate effectively, not document comprehensively. There is nothing wrong with sophisticated CASE tools in the hands of people who know how to use them, but AMDD does not depend on such tools.

Figure 4 depicts a high-level lifecycle for AMDD for the release of a system [9]. Each box represents a development activity. The initial up front modeling activity occurs during cycle/iteration 0 and includes two main sub-activities, initial requirements modeling and initial architecture modeling. The other activities – model storming, reviews, and implementation – potentially occur during any cycle, including cycle 0. The time indicated in each box represents the length of an average session: perhaps you will model for a few minutes then code for several hours.

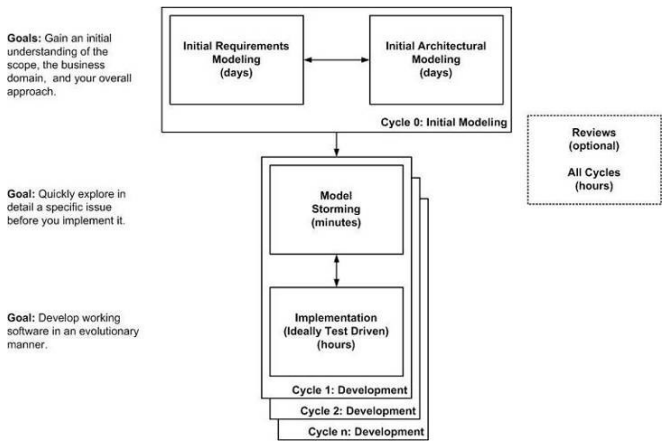

Figure 4. Taking an AMDD approach to development.

3.1 Initial Modeling

The initial modeling effort is typically performed during the first week of a long-term project. For short projects (perhaps several weeks in length) you may do this work in the first few hours and for longer projects (perhaps on the order of twelve or more months) you may decide to invest up to two weeks in this effort. You should not invest any more time than this as you run the danger of over modeling and of modeling something that contains too many problems (two weeks without the concrete feedback that implementation provides is a long time to go at risk).

Initial modeling occurs during cycle 0, the only time that an agile modeler will spend more than an hour or two at once modeling because they follow the practice *Model in Small Increments*. During cycle 0 you are likely to identify high-level usage requirements models such as a collection of use cases or scenarios; identify high-priority technical requirements and constraints; create a high-level (sparse) domain model; and draw sketches representing critical architectural aspects of your system. In later cycles both your initial requirements and your initial architectural models will need to evolve as you learn more, but for now the goal is to get something that is just barely good enough so that your team can get coding. In subsequent releases you may decide to shorten cycle 0 to several days, several hours, or even remove it completely as your situation dictates.

3.2 Model Storming

During development cycles you explore the requirements or design in greater detail, and your "model storming" sessions are often on the order of minutes. Model storming is a just-in-time (JIT) approach to modeling with a twist – you model just in time and just enough to address the issue at hand. Perhaps you will get together with a stakeholder to analyze the requirement you're currently working on, create a sketch together at a whiteboard for a few minutes, and then go back to coding. Or perhaps you and several other developers will sketch out an approach to implement a requirement, once again spending several minutes doing so. Or perhaps you and your programming pair will use a modeling tool to model in detail and then generate the code for that requirement. Model storming sessions shouldn't take more than 15 or 20 minutes, otherwise you're likely not following the AM practice *Iterate to Another Artifact* properly, and often take a few minutes at most.

It's important to understand that your initial requirements and architecture models will evolve through your detailed modeling and implementation efforts. That's perfectly natural. Depending on how light you're traveling, you may not even update the models if you kept them at all.

You may optionally choose to hold model reviews and even code inspections, but these quality assurance (QA) techniques really do seem to be obsolete with agile software development. Although many traditionalists consider model reviews to be best practices they're really "compensatory practices" that compensate for common process-oriented mistakes such as:

- Distributing your team across several locations, thereby putting you at risk that the teams are not aware of what the others are doing.
- For allowing one person or a subset of people (often specialists) to "own" the model, thereby putting you at risk that the model is of poor quality or does not reflect the work of the others on the team.
- For long feedback loops, such as a (near) serial approach to development when it can be months or even years between modeling and coding activities.

When you follow AM's practices of *Active Stakeholder Participation*, *Collective Ownership*, *Model With Others*, and *Prove it With Code* you typically avoid these problems. The high-communication and open environment enjoyed by agile modelers ensures that many people, if not everyone on the team, works with all artifacts. This ensures that many "sets of eyes" see any given model, thereby increasing the chance that mistakes are found early. The focus on producing working software ensures that the ideas captured in models are quickly put to the test – very often something will be modeled and then implemented the very same day. In these environments the value of reviews quickly disappears.

3.3. Implementation

Implementation is where your team will spend the majority of its time. During development it is quite common to model storm for several minutes and then code, following common agile implementation practices for several hours or even days. These implementation practices are:

1. **Code refactoring**. Refactoring [12] is a disciplined way to restructure code to improve its design. A code refactoring is a simple change to your code that improves its design but does not change its behavioral semantics.
2. **Database refactoring**. A database refactoring [10] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. There are different types of database refactorings. Some focus on data quality (such applying a consistent format to the values stored in a column), some focus on structural changes (such as renaming or splitting a column), whereas others focus on performance enhancements (such as introducing an index). Structural database refactorings are the most challenging because a change to the structure of your database could cause your application (or others) to crash.
3. **Test-Driven Development (TDD)**. Test-driven development (TDD) [13, 14], also known as test-first programming or test-first development, is an approach where you identify and write your tests before your write your code. There are four basic steps to TDD. First, you quickly add a test (just enough code to fail), the idea being that you should refuse to write new code unless there is a test that fails without it. The second step is to run your tests, either all or a portion of them, to see the new test fail. Third, you make a little change to

your code, just barely enough to make your code pass the tests. Next you run the tests and hopefully see them all succeed – if not you need to repeat step 3. There are several advantages of TDD. First, it ensures that you always have a 100% unit regression test suite in place, showing that your software actually works. Second, TDD enables you to refactor your code safely because you know you can find anything that you "break" via a refactoring. Third, TDD provides a way to think through detailed design issues, reducing your need for detailed modeling.

These three techniques are effectively enablers of AMDD. Refactoring helps you to maintain a quality design within your object schema over time and supports detailed changes to your design that aren't captured within your design models. Similarly database refactoring helps you to maintain a quality design within your data schema, in many ways it could be thought of as normalization after the fact. Both techniques push evolutionary design decisions into the hands of the people most qualified to make them – the people actually building the system. AMDD and TDD go hand-in-hand because they are both "think before you code" techniques. AMMD provides a way to think through big issues whereas TDD provides a way to think through detailed issues.

4. Conclusion

Modeling is a skill that all developers must gain to be effective. Agile Modeling (AM) defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. Modeling can easily become an effective and high-value activity if you choose to make it so; unfortunately many organizations choose to make it a bureaucratic and documentation-centric activity which most developers find intolerable.

The Agile Model Driven Development (AMDD) process describes an approach for applying AM in conjunction with agile implementation techniques such as Test Driven Development (TDD), code refactoring, and database refactoring. AMDD enables agile developers to think through larger issues before they dive down into the implementation details. AMDD is a valuable technique to have in your intellectual toolbox.

5. Resources

1. Beck, K., and Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA'89*, pp. 1–6.
2. Ambler, S.W. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons Publishing. 2002
3. Ambler, S.W. *An Unofficial Profile for Data Modeling Using the UML*. www.agiledata.org/essays/umlDataModelingProfile.html
4. Beck, K. *Extreme Programming Explained – Embrace Change*. Reading, MA: Addison Wesley Longman, Inc. 2000
5. Palmer, S. R. & Felsing, J. M. *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ: Prentice Hall PTR. 2002.
6. Kruchten, P. *The Rational Unified Process 2nd Edition: An Introduction*. Reading, MA: Addison Wesley Longman, Inc. 2000
7. Ambler, S.W., Nalbone, J, and Vizdos, M.J. *The Enterprise Unified Process: Extending the Rational Unified Process*. Upper Saddle River, NJ: Prentice Hall PTR. 2005.
8. Cockburn, A. *Agile Software Development*. Reading, MA: Addison Wesley Longman, Inc. 2002
9. Ambler, S.W. *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2*. New York: Cambridge University Press, 2004.

10. Ambler, S. W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York: Wiley. 2003.
11. *Model Driven Architecture (MDA) Home Page*. www.omg.org/mda/
12. Fowler, M. *Refactoring: Improving the Design of Existing Code*. Menlo Park, CA: Addison Wesley Longman. 1999.
13. Astels, D. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall. 2003.
14. Beck, K. *Test Driven Development: By Example*. Boston, MA: Addison Wesley. 2003.
15. Ambler, S.W. *Agile Model Driven Development is Good Enough*. IEEE Software, September/October 2003, 20(5), pp. 70-73.
16. Ambler, S.W. *Inclusive Modeling*. www.agilemodeling.com/essays/inclusiveModeling.htm. 2004.
17. Breen, P. *Software Craftsmanship*. Boston, MA: Addison Wesley. 2002.
18. Ambler, S.W. Just Barely Good Enough Models and Documentation? www.agilemodeling.com/essays/barelyGoodEnough.html
19. Ambler, S.W. *The Elements of UML 2.0 Style*. New York: Cambridge University Press. 2005.
20. Beck, K. *eXtreme Programming eXplained: Embrace Change.* Addison Wesley. 1999
21. Beck, K. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison Wesley. 2004

Bio:

Scott W. Ambler (www-306.ibm.com/software/rational/bios/ambler.html) is Practice Leader Agile Development with IBM's methods group and a Senior Contributing Editor with Dr. Dobb's Journal. He is founder and thought leader of the Agile Modeling (AM) (www.agilemodeling.com), Agile Data (AD) (www.agiledata.org), and Enterprise Unified Process (EUP) (www.enterpriseunifiedprocess.com) methodologies. He is (co-)author of 19 IT-related books, several of which have won industry awards.